Comput Syst Sci & Eng (1996) 2: 67–81

Computer Systems
Science & Engineering

Sadiq M. Sait, Elleithy KM, MasudulHasan•Formal synthesis of VLSI layouts from algorithmic specifications • COMPUTER SYSTEMS SCIENCE AND ENGINEERING 11 (2): 67-81 MAR 1996

Formal synthesis of VLSI layouts from algorithmic specifications

Sadiq M Sait, Khaled M Elleithy and Masud-ul-Hasan

Department of Computer Engineering, King Fahd University of Petroleum and Minerals, Dhahran-31261, Saudi Arabia

Due to advances in VLSI technology, it is possible to implement complex digital systems on a single chip. However modelling such large and complex systems at structural level is tedious and error prone. This fact has motivated the development of several high-level synthesis systems. The process consists of translating the abstract behavioural representation generally at the algorithmic level into a structural realizable representation. In this paper we present a formal approach for high-level synthesis. This formal high-level synthesis system uses μ-recursive algorithms to model the behaviour to be synthesized. These algorithms can be mathematically verified for correctness before being subjected to the task of translation. As a case study, the modelling and synthesis of VLSI layouts for matrix-matrix multipliers is discussed.

Keywords: formal synthesis, VLSI layouts, algorithmic specifications, high-level synthesis

1. INTRODUCTION

Due to advances in VLSI technology, it is possible to implement complex digital systems on a single chip. However modelling such large and complex systems at structural level is tedious and error prone. This fact has motivated the development of several high-level synthesis systems. High-level synthesis (HLS) refers to the process of synthesizing the hardware of a digital system from an abstract high-level description. The process consists of translating the abstract behavioural representation, generally at the algorithmic level, into a structural realizable representation.

The high-level synthesis problem has two main subproblems, one the extraction of structure from the behavioural description of digital systems, and two, the optimization of data flow and control flow, subject to a set of constraints (resources, timing, space, etc.). In order to alleviate the complexity of the synthesis problem restrictions are imposed. These restrictions may be on (1) the constructs used in front-end specification, (2) the structure of the hardware obtained, or (3) the class of digital circuits to be synthesized. For example, a system for a specific class of applications such as protocol processors, or systolic arrays, is simpler to design compared to one that must translate any general purpose algorithmic description to efficient hardware.

The first goal of any HLS system must be to produce a functionally correct circuit which needs no further verification after the design is completed. This requirement can be accommodated by a proper choice/design of the constructs of the algorithmic input specification language. Most existing HLS systems are based on programming languages and do not emphasize the need for formal synthesis and verification. In order to support HLS, the specification language must be simple and semantically well characterized. It must facilitate writing specifications using as few predefined objects and concepts as possible. Current hardware specification languages are far from this. In this paper we present a framework for formal HLS of digital systems. The input algorithmic specification language (ASL) is based on μ-recursive descriptions to

model the behaviour to be synthesized. These algorithmic models can be mathematically verified for correctness before being translated to hardware.

The ASL constructs include (iteration, loops, recursion, etc). Therefore it is easy to express repetitive and iterative digital systems, especially those for DSP applications such as convolution, FFT, etc. Numerical processors and algorithms involving arithmetic computation can also be modelled conveniently.

Behavioural ASL models are translated into structural level models in an intermediate form called the Realization Specification Language (RSL). In this paper we introduce a high-level synthesis system whose front-end specifications can be verified for correctness before implementation in hardware. Emphasis in this paper is on the logic/layout level cell library (backend) which is a mapping of RSL (structural intermediate form) constructs to hardware, and the translation of RSL descriptions to VLSI layouts. To illustrate modelling and synthesis, as an example we present the results of synthesizing VLSI chips of various recursive matrix-matrix multiplication algorithms.

2. PREVIOUS WORK

Up to the late 1970s, research work in the CAD area has been focused toward problems related to the lower levels of the design process. These problems are known as *physical synthesis* problems. Examples of tools developed for such tasks include timing analysers, circuit extractors, placers, routers, design rule checkers, etc.[1]. The last two decades have also seen the development of an extremely large number of tools and techniques that translate both physical and structural descriptions to silicon. Serious work on the automation of higher levels of the design process did not start until about a decade ago. In this section we briefly look at some of the reported works in the area of structural and behavioral synthesis.

2.1 RTL synthesizers

Timid efforts to describe large digital systems at the register transfer level began in the early 1960s. This was the time when CAD started being recognized as indispensable. One of the earliest reported works is that of Falkoff[2] who used APL to describe IBM System/360. Following this, special RTL languages, called Computer Hardware Description Languages (CHDL), were developed for the structural representation of digital systems. Well known amongst these are AHPL[3], CDL[4], DDL[5] and RTS[6]. The prime objective of CHDLs was to provide a set of notations which could be used to develop a well structured, unambiguous, and concise description of complex digital systems in order to improve communication among designers. There are some excellent textbooks that used these languages as a vehicle to illustrate practical design approaches[3, 4]. Soon it was realized that a description written in a CHDL can be translated automatically into a logic network[5, 7–9]. CHDLs and associated simulators and compilers demonstrated the viability of implementing complex digital systems directly from their RTL descriptions. AHPL DASys is one such synthesis system which generates logic from structural descriptions at the register transfer level (RTL), and then converts the produced logic descriptions to IC layouts[8].

2.2 High-level synthesizers

Notation to describe digital systems at even higher levels of abstraction was also developed. Most well known among these are ISPS[7], PMS[10] and VHDL[11]. Systems to produce RT-level hardware designs or structural representations from high-level description languages (or models) have been reported. These include the CMU-DA[7] and Emerald[12] systems, which produce data paths from ISPS, and the design automation assistant (DAA)[13] which generates hardware using a rule-based approach. The ADAM system of the University of Southern California[14] takes as input a high-level behavioural description in the form of a control and timing graph and generates RTL designs. All the above systems adopt a unique functional specification language whose grammars require that the target architecture be almost completely defined before using the language.

In the past few years several attempts have been made to build a top-down synthesis system which will generate hardware from high-level models. The YASC silicon compiler automatically synthesizes general cells[15]. It accepts a behavioural description as an input and transforms it into a boolean level description, which is synthesized into custom cells that are physically synthesized into layout. MIMOLA is a design system which integrates several tools for the automatic synthesis of digital systems from high-level descriptions (data flow graphs)[16]. In MIMOLA, a maximum parallel schedule is first generated, then global module allocation is performed by modelling it as an integer programming problem.

The Flamel system generates a bit-slice hardware from a Pascal program by using compiler-optimizing techniques[18]. Ideas for transforming Ada programs into silicon chips have been suggested[19, 20]. HARP is another silicon compilation system which creates an RTL description from a Fortran program[21]. It is used in conjunction with a VLSI back-end to produce mask patterns.

The first high-level synthesis system to achieve substantial commercial acceptance was the GENESIL silicon compiler[22]. GENESIL is a complete system for chip design using high-level structural elements. The heart of GENESIL consists of a set of functional blocks (PLAs, ROMs, RAMs, etc.). These blocks can be selected and tailored by users to their specific needs. The control structure can be specified as a state machine and the logic is entered as a truth table.

2.3 Formal high-level synthesizers

None of the above systems include features and facilities

to verify the correctness of the design before synthesis. In formal high-level synthesis a mathematical framework is used to assist and verify characteristics at the algorithmic level before the synthesis step. The mathematical framework is the basis for producing correct-by-construction architectures that need no further verification. An approach for transforming a system described in the form of linear recursive equations to digital systems has been presented[23]. Another approach based on temporal logic as a mathematical framework to produce *n*-dimensional systolic arrays[24, 25], and a formal language (ASL) based on μ-recursive algorithms hasve also been presented[26, 27] in the literature.

A formal synthesis system that uses a tool called Lambda has been introduced by Bombana et al.[28]. This Lambda tool was integrated into the *Itallel* design system using EDIF and VHDL. Another work on formal synthesis for self-timed circuit design based on a compact event model was introduced by Kishnevsky *et al.*[29]. In this tool, the formal synthesis procedure is considered as a set of equivalent transformations from some initial specifications. This approach, however, can only be used to produce self-timed circuits.

The formal framework discussed in this paper accepts high-level descriptions in ASL and produces structural descriptions of the intended design in RSL. Figure 1 illustrates all the processing steps of this synthesis procedure. The procedure is not dedicated to any application specific domain or architecture. However, for a class of digital systems such as DSP circuits, efficient VLSI layouts are produced.

3. ASL PRIMITIVES

The ASL is based on μ-recursive functions as a framework. Recursion is a natural way for defining *any* computable function, for example, the set of natural numbers

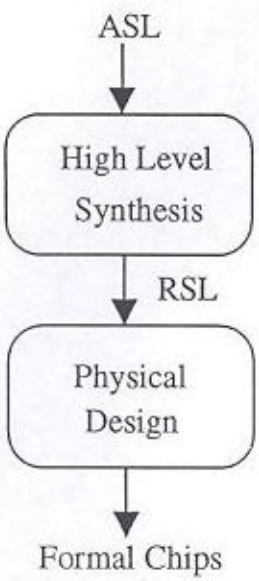

Figure 1 Flow chart illustrating the steps of the formal synthesis procedure

N can be defined starting from zero and adding one recursively. This can be stated formally as follows:

$$N(0) = 0$$
$$N(X+1) = N(X) + 1$$

The importance of recursion is not limited to specifying functions, but it can be used to prove most properties of algorithms by inference, which is called proof by *induction*. In ASL, a given algorithm is represented using a limited number of constructs. Although the constructs are primitive, complex constructs can be developed through a cell library. ASL consists of *initial functions* and *operations*. The initial functions are very primitive functions. The following three initial functions are used:

1. The *zero function* ($\{\xi()\}$) which returns zero.
2. The *projection function* ($\eta_i^n(\arg_1, \arg_1,...,\arg_n)$) which chooses an argument *i* from *n* arguments.
3. The *successor function* ($\lambda(n)$) which increments the input by one.

The three operations that are used to construct larger functions from smaller ones are:

1. *Composition.*
2. *Recursion.*
3. *μ-recursion.*

The complete details of ASL are given elsewhere[26, 27]. Some of the characteristics of the ASL are:

1. It is simple and semantically well characterized.
2. It permits writing specifications using a few predefined objects and concepts.
3. It is a suitable tool for formal synthesis.
4. It supports a hierarchical synthesis methodology.
5. It supports formal verification of the liveness and safeness properties of a design.
6. It is complete.

3.1 Structural synthesis from ASL

A formal mapping methodology is developed to transform an algorithm represented in ASL to a specific realization language termed RSL. RSL specifies the components and connectivity of a digital architecture that realizes the algorithm. Every construct in ASL has an isomorphic representation in RSL, which is the basis for the automated transformation. Details of the transformation algorithms for mapping from ASL to RSL can be found elsewhere[26, 27, 30] (also see Appendix A). In the following section we give a brief overview of RSL.

4. OVERVIEW OF RSL

The system under consideration accepts ASL as an input and produces an intermediate form RSL which acts as the

specification for the back-end. This back-end produces a VLSI layout from the RSL specification. The specification would have three initial functions, and three operations, which can be applied in a certain sequence to express any computable function or hardware. In order to build such a system, it is required that the intermediate representation RSL be transformed into layouts. To achieve this, hardware of each initial function is synthesized and stored in a cell library. Details of the RSL language and the transformation procedure from ASL to RSL are given in Appendix A.

The approach used for the implementation of RSL is illustrated in Figure 2[31]. It shows that the RSL specifications of a given circuit is translated into logic level models. The formal cell library contains logic/gate level models of all the basic functions of RSL. Modularity and design extendibility to any desired word length are taken into consideration while designing basic cells. This logic level model of the given circuit is then simulated using a gate level simulator. The functionally correct model is then given to the input of the physical design sub-system which generates the final VLSI layout. This physical design system uses a cell library of pre-designed standard cells.

4.1 Formal logic cell library design

As mentioned earlier, any synchronous digital system can be expressed using the basic functions and constructs of ASL, viz, *zero, projection, successor, composition, recursion* and μ-*recursion*. Corresponding to each construct in ASL is a construct in the realization specific language (RSL) that must be mapped to hardware equivalent modules. For each RSL primitive, the library contains a logical description (at the gate/transistor level). Similar to the construction of larger functions at the ASL level, larger logic modules are constructed using cells of initial primitives in the library. The synthesized logic circuits are stored in the library for later use. Therefore, for ease of expandibility, they are made modular so that they can be easily connected to build cells of larger functions. Also the design accommodates varying word lengths. For example, an n-bit successor function is easily made by cascading n 1-bit successor units. Thus the cell library designed consists of modular building blocks of logic level macros with the above characteristics. Each primitive logic module is carefully synthesized. The approach used for making the formal cell library is shown in Figure 3[31].

For verification by simulation, the basic functions, which are *zero, projection* and *successor*, are modelled at the logic level and expressed using a *netlist* language[32].

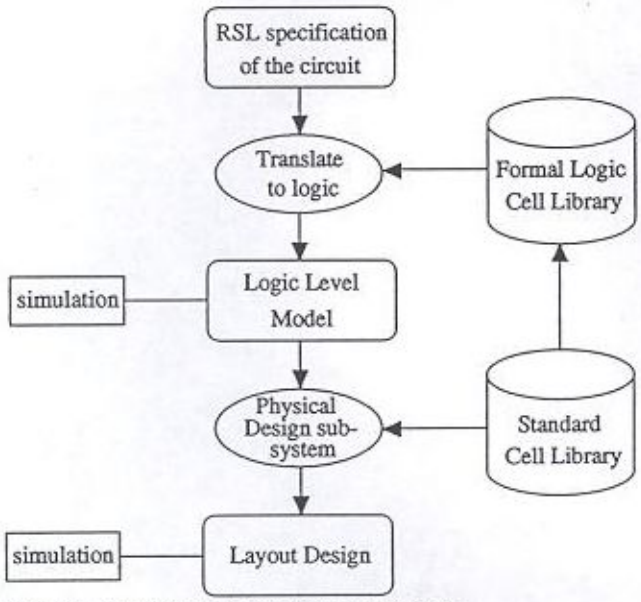

Figure 2 Flowchart for the implementation of RSL

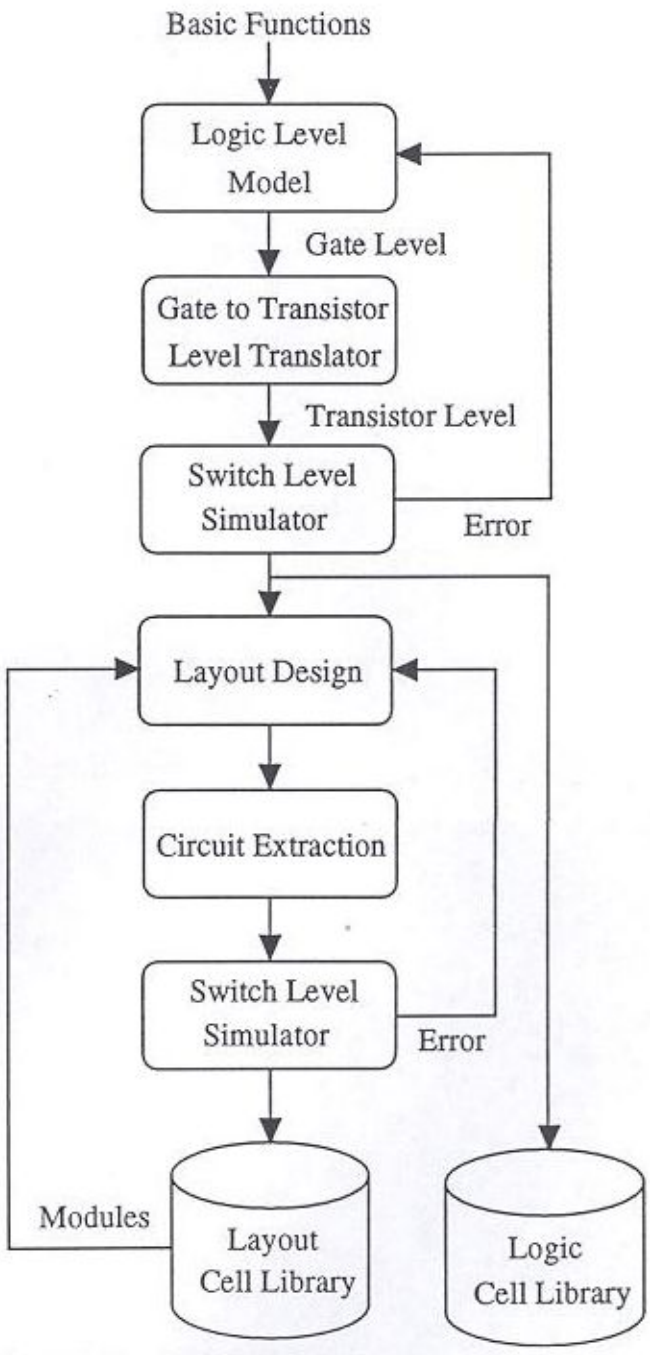

Figure 3 Design methodology of the cell library

The logic level netlist is then translated into a transistor-level circuit using a tool called **netlist** and this transistor-level circuit is then translated into a binary file using a translation tool called **presim**[32]. This binary file is required for simulation by **rnl**, a switch level simulator[32]. This simulation will verify the correctness of translation. Layouts of these functions are then made using a layout synthesis system such as OASIS and Magic[33-35]. Circuit from layouts are extracted and simulated to verify the functional correctness of layouts. These stored initial functions are used in the definition and synthesis of larger functions.

Clocking strategy

A two-phase non-overlapping clocking scheme is used in the design of all the cells (or sub-circuits). Input is loaded during ϕ_1 and the output is obtained during ϕ_2. The large modules are made by using the primitive functions. Every cell has an input *Control* and an output *Ready*. The input signal *Control* is used to signal the start of operation and the output line *Ready* is used to indicate that the circuit has finished its operation. The operation of individual elements is data driven. The cells are connected in such a manner that the *Ready* of one is connected to the *Control* of the next cell. The cell starts its function when its *Control* gets some signal from the *Ready* of the previous cell.

5. LOGIC SYNTHESIS

We now describe the implementation of the three primitive functions which are used to implement larger functions.

5.1 Zero function

The *zero* function returns the value *zero* and has no arguments. A register is used to implement the function which is initialized with the value *zero* ($\xi()$).

5.1.1 Projection function

The *projection* function is used to choose an argument i from n arguments. A multiplexer is used to perform the *projection* function. $N + 1$ registers are used to store the arguments. An input line *Control* is used to initiate the multiplexer operation, and an output line *Ready* is used to indicate that the circuit's operation has been completed.

Figure 4 shows the hardware of *projection* function. It is a simple 2-to-1 line multiplexer constructed using three transmission gates. There is an input selection line *sel* to select one of the two inputs i.e. *in0* and *in1*. The output *out* is obtained by the enable line *Control*. Larger units are similarly constructed using only transmission gates.

5.1.2 Successor function

The *successor* function is basically an incrementer that takes an input n and produces the output $n + 1$. An adder and two (dynamic) registers are used to model the successor function in RSL. One of the registers stores the argument n and the other the value *one*. The operation is executed in one clock cycle. The block diagram of a 1-bit successor cell is shown in Figure 5.

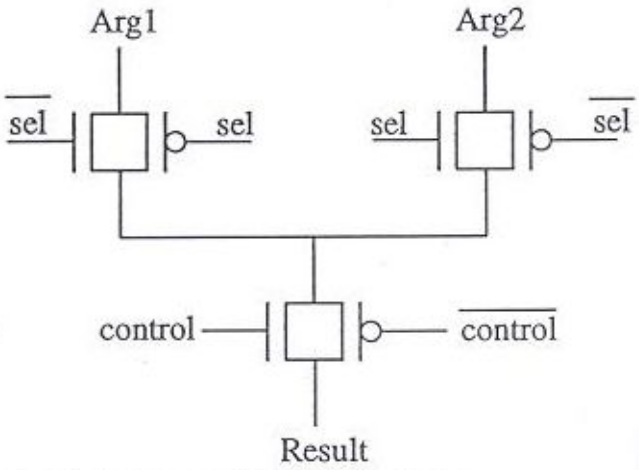

Figure 4 Hardware model of the *projection* function

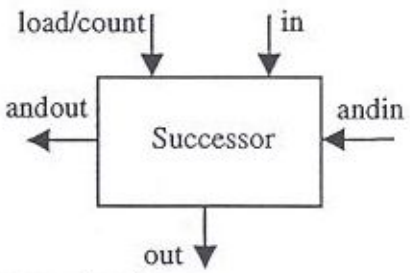

Figure 5 Successor function

The successor unit in our library is designed as an incrementer and can also be used as an up-counter by feeding back the output to one of its two inputs. *Load/Count* control inputs are available to accomplish the necessary function. Input is loaded during ϕ_1 and the output is obtained during ϕ_2. The cascading of successor units can be done by connecting the *andout* output of a unit to the *andin* input of the adjacent unit. The hardware model of the successor function is shown in Figure 6.

6. FORMAL CELLS OF LARGER FUNCTIONS

Larger functions are made by using the basic functions. For example, the successor function can be used to build an adder, and adder can be used to build a multiplier. The adder and multiplier can be used to synthesize an inner-product cell, which can then be used to implement say a matrix-matrix multiplier. We will first discuss the details of the *Add* unit.

6.1 Add unit

The *Add* unit is used to add two arguments. Addition of n and m is modelled in ASL by incrementing n times the argument m.

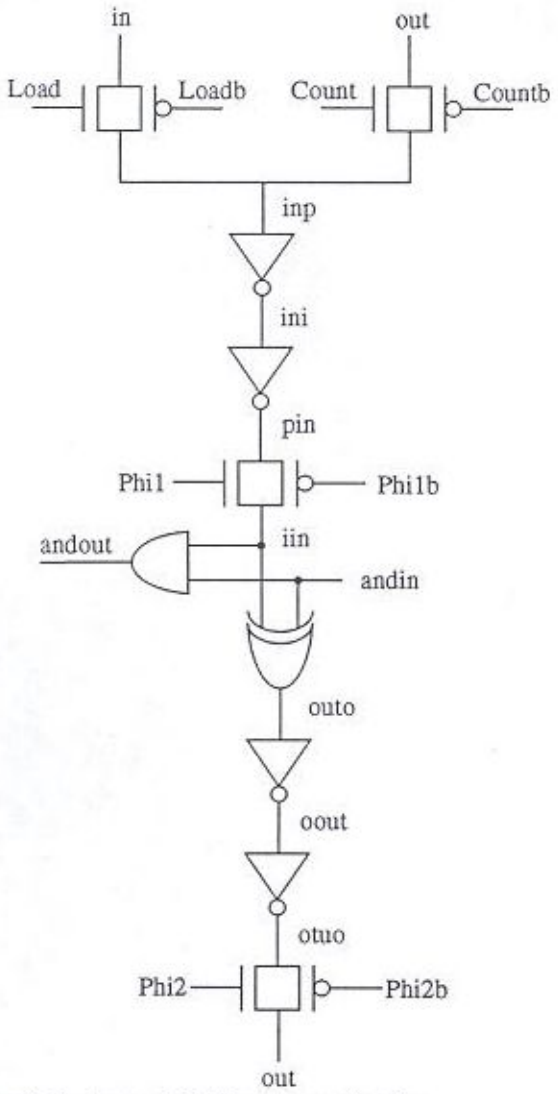

Figure 6 Hardware model of the *successor* function

The direct block diagram representation of the *Add* function in RSL is shown in Figure 7. It is implemented using two *successor* functions and a comparator. Addition is done in a recursive fashion. The *successor* function which is connected to one of the inputs of comparator is initialized with an input of *zero*, at the same time the other *successor* function is loaded with one of the two arguments to be added, say n. The other argument, say m of the adder, is given to the second input of the comparator. The output *Ready* of the comparator becomes high only when both of its inputs become equal. Both of the successors start incrementing simultaneously and increment m times. When the output value of the successor connected with the comparator reaches the value equal to m, *Ready* goes high and both the successors stop incrementing as the *Ready* is connected to the *Control* of both the successors. Evidently the successor function loaded with the initial value of n gives the final result as $m + n$ (see Appendix B).

6.2 Pro unit

In ASL, multiplication of two number n and m is modelled as addition of n to itself m times. The unit *Pro* is used to multiply two arguments.

The direct block diagram representation of the *Pro* function in RSL is shown in Figure 8. It is implemented using one successor function, one comparator, one Add unit and some logic gates as control circuitry. The multiplication is done in a recursive manner. Suppose m is to be multiplied with n. The successor function which is connected to one of the comparator's input is initialized with an input of *zero*. The argument m is given to the second input of the comparator. The unit *Add* is loaded with two arguments *zero* and n. The unit *Add* adds n to an empty register, m times, giving the final result $m \times n$ (see Appendix B).

6.3 Inner-Product unit

The *inner-product* unit is used to multiply two arguments and add the result with the third argument. Suppose a, b and c are three inputs to the inner-product unit and d is the output. The function of the cell is given by:

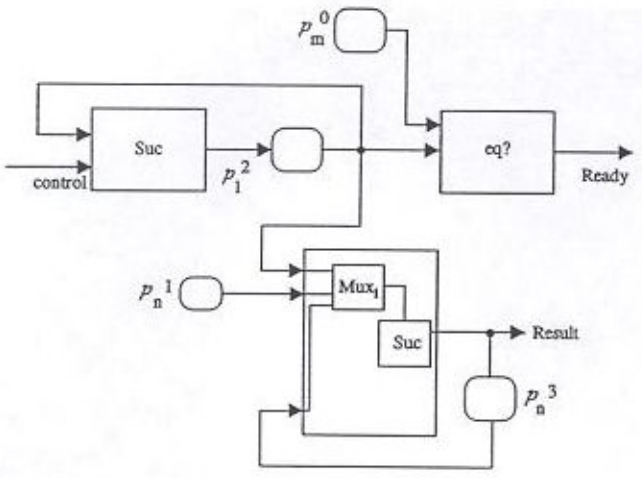

Figure 7 RSL architecture of the unit *Add*

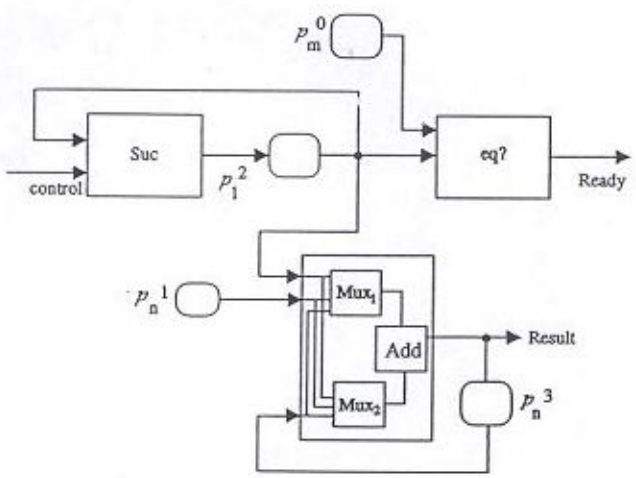

Figure 8 RSL architecture of the unit *Pro*

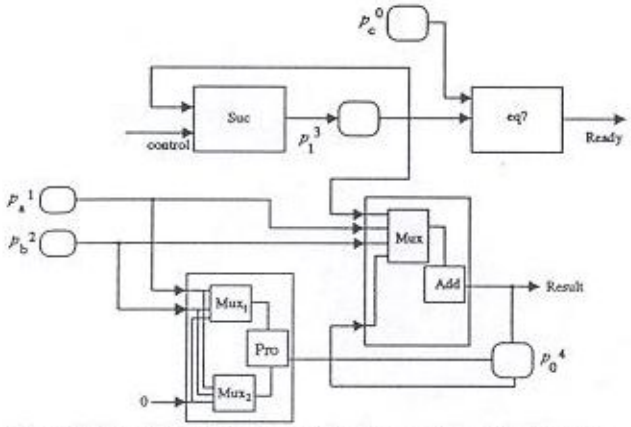

Figure 9 Thre RSL architecture of the *inner-product* cell. Mux represents the projection unit. Details of *Add* and *Pro* are given in Figures 7 and 8

$$d = a \times b + c$$

The direct block diagram representation of *inner-product* unit in RSL is shown in Figure 9. It is implemented using one *Add* circuit, and one *Pro* circuit, and some logic gates. The *Pro* multiplies two arguments in a recursive manner and the *Add* unit adds the third argument to its result (see Appendix B).

7. LAYOUT SYNTHESIS

In the previous section, we discussed the development of the logic level formal cell library. This section deals with the layout design methodology of the cells stored in the formal cell layout library, as well as the layouts of the formal cells which use the basic cells in their definition.

Cell layout design methodology

In a cell based VLSI layout synthesis system, the heart of the physical design unit is the cell library. The approach for such a synthesis system is illustrated in Figure 10[31]. The logic level model, using a formal logic cell library, of the given circuit is fed to the physical design system. Two sub-systems of the physical design system are used. The first sub-system is used for placement and routing using a standard cell library. It places the cells and performs the global and detailed routing in a plane to minimize the layout area. The output of the first sub-system is used by the other sub-system. It assembles the final physical layout. The circuit is then extracted and simulated to verify that the hardware description of the primitive function under design performs the intended function.

Physical design sub-system

Layouts are made using the layout design environment Open Architecture Silicon Implementation Software (**OASIS**)[33, 34]. A number of design tools are integrated into the OASIS system. It is a cell-based system for IC design. The tools integrated into the OASIS system have been developed to automatically translate high-level

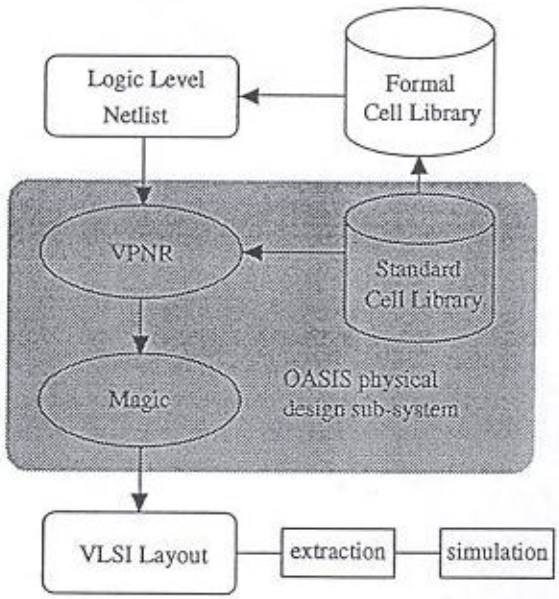

Figure 10 Approch for cell-based VLSI layout synthesis system

descriptions or netlist similar to those designed for RSL, into testable physical layouts, using predesigned standard cells.

One of the premises of the OASIS system is the modularity of the software. New, improved algorithms can be easily substituted in place of the old ones. The entire system is controlled with a single data flow supervisor program to assure data consistency is maintained at all stages of the design. The data flow supervisor is template-driven; the templates used in OASIS can be easily expanded to support additional software tools, thus providing the desired openness of the system.

Standard cell approach

The layouts produced with the OASIS system utilize standard cells of a uniform height. The cells are placed in an array of horizontal rows and all interconnection of signal nets are made by channel routing in the space between the adjacent rows. All connections of the power nets, *VDD* and *GND*, are made by abutting the cell horizontally. Signal nets connecting cells belonging to non-adjacent rows cross the intervening rows of the cells by utilizing feed-through pins (vertical strips of metal running from the top to bottom of a cell) built into some cells, or by inserting feed-through cells (cells consisting solely of a single feed-through) whenever appropriate. A set of scalable CMOS cells compatible with the 2μ SCMOS technology are used. Cell signal input and output ports are made in the second layer of metal, while the power net (*VDD* and *GND*) ports are made in the first layer of metal.

The *VDD* and *GND* power nets from each row of cells are connected together using power rails running vertically through the entire height of the layout.

Placement

The placement and routing in OASIS is done by a layout

subsystem of OASIS called *VPNR* (Vanilla Place aNd Route). The tools comprising VPNR create physical layouts automatically from netlist descriptions of logic circuits, using a library of pre-designed standard cells of uniform height. The goal of VPNR is to place cells and perform global and detailed routing of interconnections in a plane so as to minimize the layout area. VPNR is usually invoked at the end of the entire design process after the design has been simulated and verified to implement the desired functionality. Occasionally, the layout may be generated in the early stages of the design to obtain an estimate of the area taken up by the circuit. VPNR employs placement and global routing algorithms based upon the quadrisection paradigm[33]. The combined placement and global routing program receives a netlist of standard cells, partitions it into four quadrants (top-left, top-right, bottom-left and bottom-right), and processes each quadrant in the same manner until each quadrant contains one cell row in the vertical direction. Partitioning is accompanied and directed by approximate global routing.

Routing

After the positions of the cells in rows are determined, the algorithm constructs a minimum spanning tree for each net, finds the exact crossing locations for nets that need to cross the cell rows, insert feed-through cells if necessary, and assigns sub-nets in channels. The nets are processed sequentially, and the routing of each net takes into account all nets routed previously. After all nets are routed, the global router prepares the data for detailed channel routing. The channel router determines where to put the wires so the resulting layout occupies the least amount of area. VPNR provides a choice of two channel routers: a greedy router[36] and a left edge based router with channel compaction[37].

Simulation and verification

The entire purpose of this phase is to verify that the hardware description of the basic cells under design performs the intended function. The layouts made by Magic can be extracted and simulated using a simulation tool called **irsim**[17, 35]. There is, however, no need to verify the correctness of the entire design, which is supposed to be correct by construction.

Layouts of the different units discussed in the previous sections are synthesized using layout design environment presented above[33, 34]. The VLSI layouts of cells are implemented using SCMOS technology.

7.1 Layouts of formal matrix-matrix multipliers

An example of formal matrix-matrix multipliers is presented as an application of the cell library[27]. Three different types of architectures, i.e. (1) simultaneous recursion, (2) recursion with respect to several variables and (3) recursion with fixed number of nestings, are used. Each architecture accepts two matrices as input, and produces a third matrix as output. The ASL and RSL specifications of all the three types of multiplier are given elsewhere[26, 27]. In this section we first present the high-level subroutine of a matrix-matrix multiplier which uses simultaneous recursion on *inner-product units*. The multiplication is done recursively as described below. Suppose A and B are the two input matrices and C is the output matrix.

```
matrix-multiplication (A, B, C)
begin
  for i = 1 to n
    for j = 1 to n
    begin
    C_{i,j,0} = 0
      for k = 1 to n
        C_{i,j,k} = C_{i,j,k-1} + A_{i,k} * B_{k,j}
      next k
    end
  next i
end
```

The above algorithmic description is translated into an ASL description as follows:

$$C_{1,1}(A_{1,k}, B_{1,j}, 0) = \xi()$$
$$\ldots\ldots\ldots\ldots$$
$$C_{n,n}(A_{n,k}, B_{n,j}, 0) = \xi()$$
$$\ldots\ldots\ldots\ldots$$
$$C_{1,1}(A_{1,k}, B_{1,j}, K) = inner-product(A_{1,k}, B_{1,j}, C_{1,1}(A_{1,k}, B_{1,j}, K-1))$$
$$\ldots\ldots\ldots\ldots$$
$$C_{n,n}(A_{n,k}, B_{k,n}, K) = inner-product(A_{n,k}, B_{k,n}, C_{n,n}(A_{n,k}, B_{k,n}, K-1))$$

The architecture of the formal matrix-matrix multiplier, using **simultaneous recursion,** consists of n^2 inner-product cells. Figure 11 shows the hardware model. This type of architecture gives output in a serial manner. Figure 12 shows its VLSI layout.

The formal matrix-matrix multiplier using **recursion with respect to several variables**, is implemented by recursion construct on *inner-product* units. The architecture consists of n multiplication units and 1 adder unit. Figure 13 shows the hardware model of this type of *matrix-matrix multiplier*. Figure 14 shows the VLSI layout of a matrix-matrix multiplier using recursion with respect to several variables.

The architecture of formal matrix-matrix multiplier using **fixed nesting recursion** consists of n inner-product units. Figure 15 shows the hardware model of this type of *matrix-matrix multiplier* and Figure 16 shows its VLSI layout. The ASL and RSL descriptions of the above two matrix-matrix multipliers are given elsewhere[26, 27].

Table 1 shows the number of devices and layout area of various designs for an 8-bit data bus. Table 2 shows the number of clock cycles required by these units. It can be observed that the area and time are high as compared to the that of designs by non-formal methods. It is the price paid for the functionally correct hardware made by formal techniques.

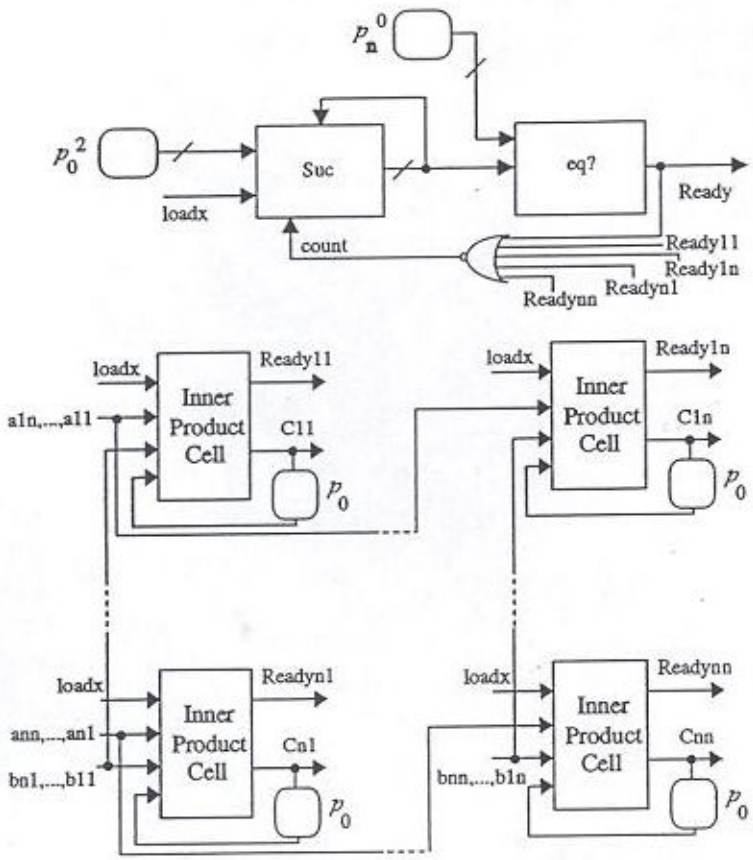

Figure 11 Hardware model of a *matrix-matrix multiplier* using simultaneous recursion

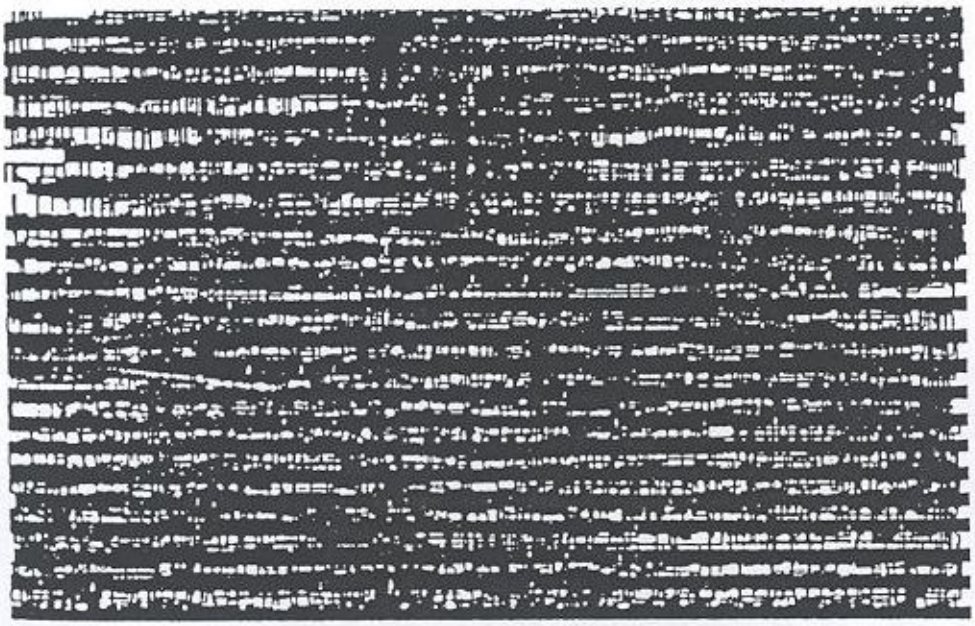

Figure 12 VLSI layout of an 8-bit *matrix-matrix multiplier* using simultaneous recursion

8. CONCLUSION

In this paper, we present the structure of a formal high-level synthesis system. A formal cell library to support high-level synthesis of digital systems was also presented. The cell library introduced consists of both logic level and layout level cells directly mapped from primitives. It supports a hierarchical design methodology and can be automatically translated to layouts.

Large digital systems can be easily modelled using the formal primitives both at the ASL level, and also at the RSL level. An automated procedure is used to transform ASL representation into a specific realization. The realization format is based on representing architectures using

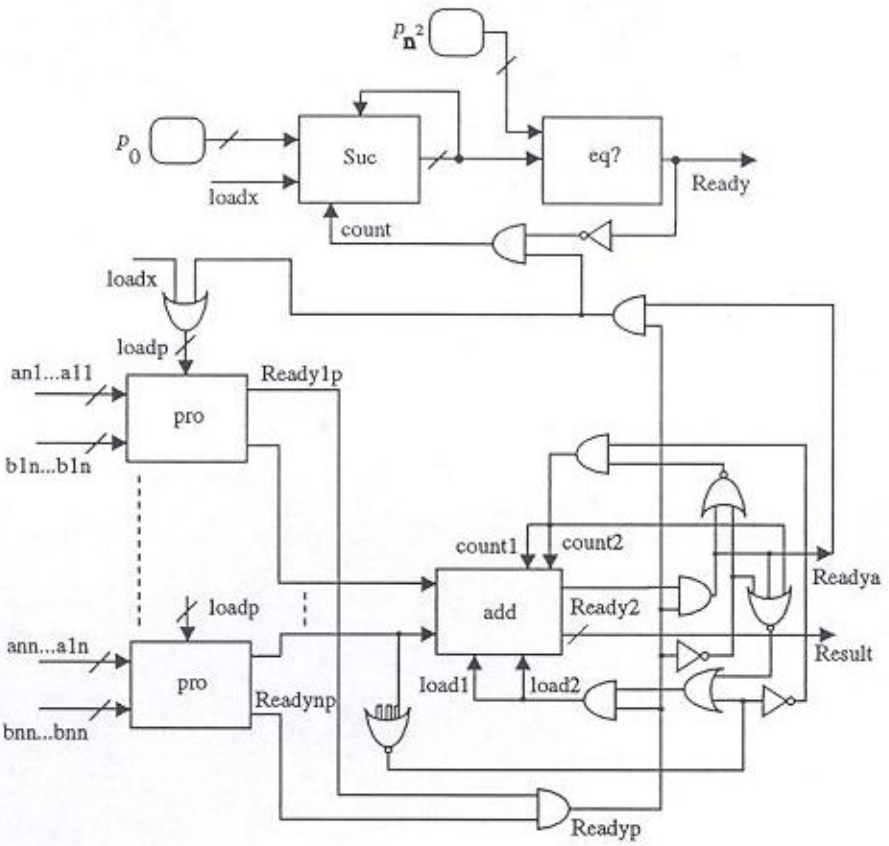

Figure 13 Hardware model of a *matrix-matrix multiplier* using recursion with respect to several variables

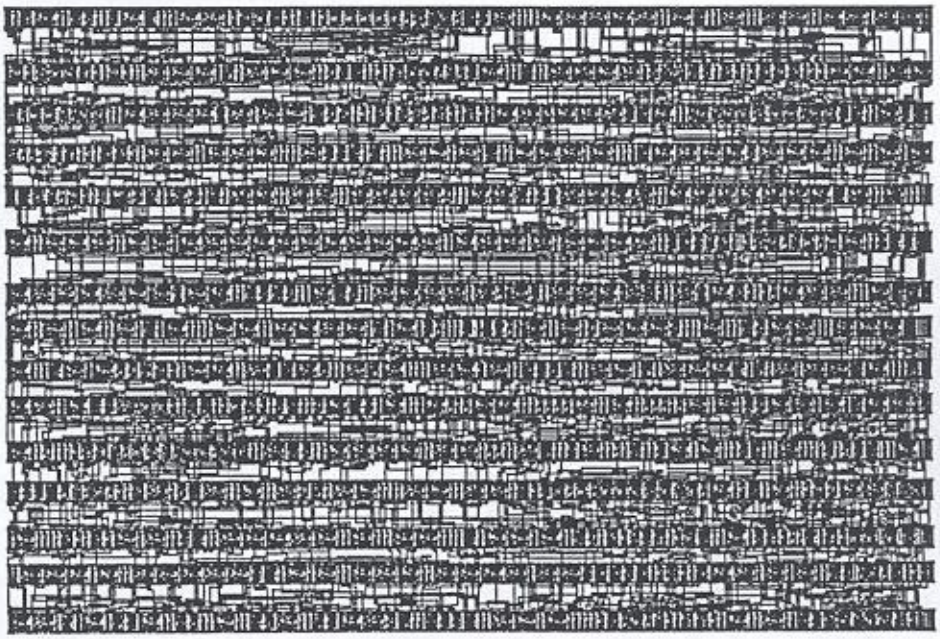

Figure 14 VLSI layout of an 8-bit *matrix-matrix multiplier* using recursion with respect to several variables

RSL. Algorithms of linear order complexity are used for transformation from ASL to RSL. Designs expressed using the ASL primitives can be formally verified for correctness before synthesis. The lower level verifications such as correctness of the logic circuits, and correctness at the transistor/layout level rely on simulators.

Extraction of control and data paths of digital systems expressed in ASL is easy. The execution of various functions is based on data flow. Interaction between various units is based on a simple protocol.

A number of digital systems were modelled at the ASL level, translated to RSL and their VLSI layouts synthesized using standard-cell design methodology. As an example the design of matrix-matrix multipliers were pre-

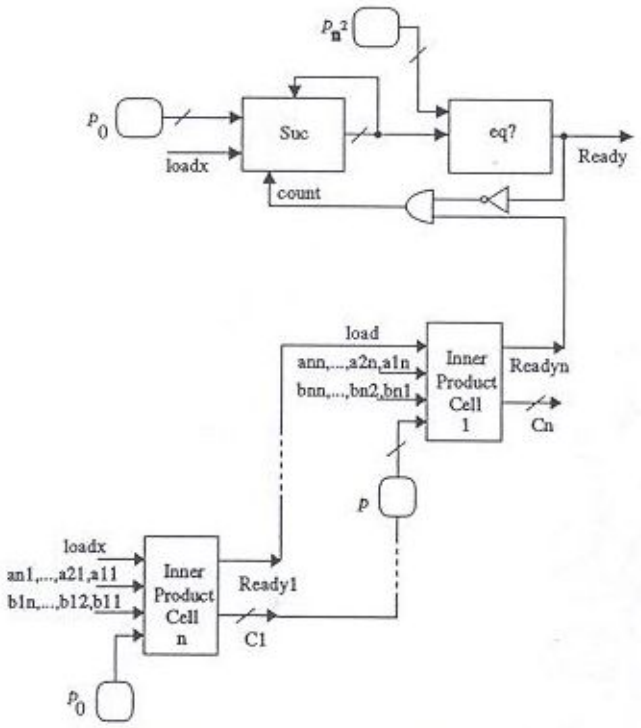

Figure 15 Hardware model of a *matrix-matrix multiplier* usingfixed nesting recursion

sented in this paper. The system can be used to model and synthesize any synchronous digital system, and is most efficient for arithmetic and DSP intensive applications.

ACKNOWLEDGMENTS

We acknowledge King Fahd University of Petroleum and Minerals for all support provided for this work.

Table 1 Number of devices and layout area of 8-bit units

Unit	No. of devices	Area (λ^2)	Other units used
counter	336	288320	successor
Add	974	739480	incrementor
Pro	1674	1284376	Add, Incrementor
inner-product	2720	2244528	Add,Pro,Incrementor
multiplier1	12060	12325000	inner-product
multiplier2	5864	4218240	Add, Pro
multiplier3	7244	5279400	inner-product

APPENDIX A: REALIZATION SPECIFICATION LANGUAGE (RSL)

Description of the Language

The realization introduced here is described at the architectural level. The specification describes two elements of a circuit: the *components* and the *connectivity*. The physical parameters of the circuit are not addressed at this level of specification. The timing is not expressed explicitly, but implicitly in terms of registers. The symbol ρ is used to represent a register. When a register is initialized to a certain value β, it is represented as β_ρ.

1. If *unit* is a certain component that is used in the system, and *inp* is any input for *unit*, then we can use this input explicitly using the following syntax:

   $unit_{inp}$

   Also an output *out* of *unit* can be used as follows:

   $unit^{out}$

2. The identifiers starting with upper case letters are

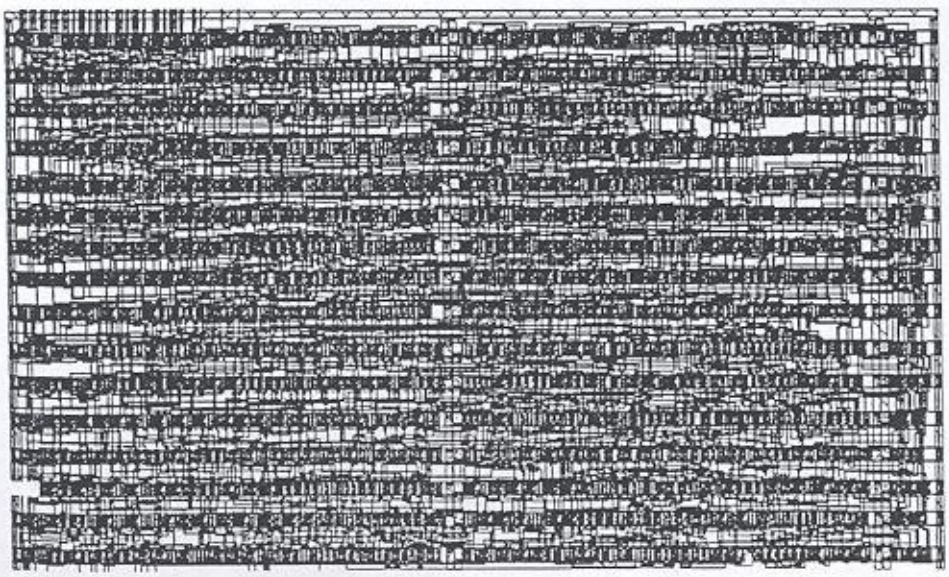

Figure 16 VLSI layout of an 8-bit *matrix-matrix multiplier* using fixed nesting recursion

Table 2 Number of clock cycles required by the 8-bit units before the final results are obtained

Unit	No. of Arguments	Arguments	Time (clock cycle)
counter	1	n	n
Add	2	m, n	max(m,n)
Pro	2	m, n	m× n
inner-product	3	a, b, c	(a×b) + max(a×b, c)
multiplier1	$2(n \times n)$matrices	$A^{n\times n}$, $B^{n\times n}$	$max(C_{11}, \cdots, C_{nn})$ where $C_{11} = (a_{11} \times b_{11} + a_{12} \times b_{21}) + max(a_{11} \times b_{11}, a_{12} \times b_{21})$
multiplier2	$2(n \times n)$matrices	$A^{n\times n}$, $B^{n\times n}$	$(C_{11} + \cdots + C_{nn})$ where $C_{11} = 2max(a_{11} \times b_{11}, a_{12} \times b_{21})$
multiplier3	$2(n \times n)$matrices	$A^{n\times n}$, $B^{n\times n}$	$(C_{11} + \cdots + C_{nn})$ where $C_{11} = (a_{11} \times b_{11}) + (a_{12} \times b_{21}) + max(a_{11} \times b_{11}, a_{12} \times b_{21})$

used to represent units that will be further expressed at lower levels and also to represent temporary variables used in the computation. While the identifiers starting with lower case letters are used to represent basic functions as well as constants.

3. When a register α is initialized to a value β, then this is expressed as ρ^{α}_{β}. For example, the expression $\rho^{1}{}_{2}suc(I)$ means that *suc* unit has the register I connected to its input, this register is initialized with the value 2. The *Init* statement is an RSL expression that can be used separately to express that register α is initialized to value β as follows:

$$Init(\alpha, \beta)$$

Another version of *Init* can be used to denote the initialization of more than one register at the same time; this is represented as:

$$Initp(\alpha_1, \beta_1; \alpha_2, \beta_2; \cdots; \alpha_n \beta_n)$$

The above statement means that n registers are initialized in parallel such that register α_1 is initialized with the value β_1, register α_2 is initialized with the value β_2 and so on. It is assumed that all registers are synchronized using a global clock.

4. Constant values are expressed using registers. If it is required to have a constant value c in the circuit, then this is represented as α_c.

The representation of functional units and the control section is presented in the following section.

Transformation from ASL to RSL

The transformation approach is based on using a one-to-one mapping procedure. The algorithm takes an ASL representation and transforms each ASL construct into an equivalent RSL representation.

1. *Zero* function returns the value and has no arguments. A register is used to realize the function which is initialized with the value zero. The RSL representation is as follows:

$$Result = zero$$
$$zero^{Ready} = zero_{Control}$$

2. *Projection* function is used to choose an argument i from n arguments. A $(k, \lceil logk \rceil)$ multiplexer is used to perform the projection function. An input line *Control* is used to determine the start of the multiplexer operation, and an output line *Ready* is used to indicate that the circuit has finished its operation. This is represented in RSL as follows:

$$Result = mux(arg_1, \cdots, arg_k \# i)$$
$$mux^{Ready} = mux_{Control}$$

3. *Successor* function is used to increment the input argument by one. An adder is used to perform the successor function. A control input signal and a ready output signal are used to determine the start and end of the operation, respectively, as in the projection function. This is represented by the following RSL code:

$$Result = suc(n)$$
$$suc^{Ready} = suc_{Control}$$

4. Composition function representation in RSL is as follows:

$$Result = Comp(x_1, \cdots, x_m \# y)$$
$$Comp^{Ready} = And(x_1{}^{Ready}, \cdots, x_n{}^{Ready})$$

Notice that *And* represents the logical AND, and it is considered as a Basic function and *Comp* is function z in the recursion construct.

5. *Recursion* function is represented in RSL as follows:

$$Result = eq?(arg_1, arg_2)$$

The RSL representation of the circuit for recursion is as follows:

$$Initp(0, m; 1, arg_1; 2, arg_2; \cdots; n, arg_n) \quad (A1)$$

$$suc_{control} = g(\overline{arg})^{ready} \quad (A2)$$

$$I = \rho_1^{n+1}\, suc(I) \quad (A3)$$

$$Ready = eq?(I, m) \quad (A4)$$

$$Result = comp(arg, I, \rho_g^{n+1}\, Result\#y) \quad (A5)$$

The first equation indicates that n+1 registers are initialized by the arguments, n of them with the arguments of the y component and a register for the constant m. The second equation means that the unit *suc*, which is a basic function, has its input control connected to the ready output of the unit computing $g(\bar{n})$ to be sure that I is not incremented until $g(\bar{n})$ is computed. The third equation is used to represent the fact that I is incremented every clock cycle using the *suc* unit, and I is initialized to the value 1 using the register number n+1. The fourth equation determines the end of the operation when I reaches the value m. The fifth equation means that y has n inputs from the n registers, one output from register number n+2 which is initialized to the value $g(\bar{n})$, and an input I which is the output of the *suc* unit.

6. μ-*Recursion* function is represented in RSL as follows:

$$Initp(o, m; 1, arg_1; 2, arg_2; \cdots; n, arg_n)$$
$$Result = \rho_0 suc(Result)$$
$$Ready = eq?(0, g(\overline{arg}, Result))$$

The first equation is used to initialize the n+1 register with y arguments and a zero value. The second equation is used to indicate that the unit *suc* has an input from a register initialized with a value 0 and the output of the *suc* is the *Result. The* third statement is used for comparing *suc* output with m.

From the previous transformation method we see that each construct in ASL has an isomorphic representation in RSL. The equivalence between ASL and RSL represents the base for the automatic transformation procedure for any algorithm specified using ASL to RSL.

APPENDIX B: FORMAL DESCRIPTION OF DIFFERENT UNITS

ASL Representation

- Multiplication unit code:

$$pro(n,0) = \xi()$$
$$add_{13}(arg_1, arg_2, arg_3) = add(\eta^3{}_1(arg_1, arg_2, arg_3), \eta^3{}_3(arg_1, arg_2, arg_3))$$
$$pro(n, m+1) = add_{13}(n, m, pro(n, m))$$

- Addition unit code:

$$add(n, 0) = \eta^1{}_1(n)$$
$$Q(arg_1, arg_2, arg_3) = \lambda(\eta^3{}_3(arg_1, arg_2, arg_3))$$
$$add(n, m+1) = Q(n, m, add(n,m))$$

- Inner-product cell code:

$$inner\_product(a, b, 0) = pro(\eta^3{}_1(a, b, 0), \eta^3{}_2(a, b, 0))$$
$$add\_one(arg_1, arg_2, arg_3, arg_4) = \lambda(\eta^4{}_4(arg_1, arg_2, arg_3, arg_4))$$
$$inner\_product(a, b, c+1) = add\_one(a, b, c, inner\_product(a, b, c))$$

RSL Representation

The RSL specification is divided into three parts as the ASL specification.

- The multiplication unit code:

$$Result_1 = zero \mid Rule_1 \quad (B1)$$
$$Result_2 = mux_1(arg_1, arg_2, arg_3\#1) \mid Rule_2 \quad (B2)$$
$$Result_3 = mux_2(arg_1, arg_2, arg_3\#3) \mid Rule_2 \quad (B3)$$
$$Result_4 = add_{13}(Result_2, Result_3\#add) \mid Rule_4 \quad (B4)$$
$$Initp(0, m; 1, n) \mid Rule_{5-1} \quad (B5)$$
$$suc_{Control} = zero^{Ready} \mid Rule_{5-2} \quad (B6)$$
$$I = \rho^2{}_1 suc(I) \mid Rule_{5-3} \quad (B7)$$
$$Ready = eq?(I, m) \mid Rule_{5-4} \quad (B8)$$
$$Result = pro(n, I, \rho^3{}_{Result_1}\#add_{13}) \mid Rule_{5-5} \quad (B9)$$

Comparing the ASL and RSL specifications we see that statement 1 in RSL is corresponding to the first statement in ASL and is obtained using rule 1 in the transformation procedure (explained in Appendix A), statements 2–4 correspond to the second statement in ASL and are obtained by applying rule 5 twice and rule 4 once, and statements 5–8 correspond to the third ASL statement and are obtained by applying rule 5 once.

- The addition unit code:

$$Result_1 = mux_1(arg_1\#1) \mid Rule_1 \quad (B10)$$
$$Result_2 = mux_2(arg_1, arg_2, arg_3\#3) \mid Rule_2 \quad (B11)$$
$$Result_3 = Q(Result_2\#suc) \mid Rule_7 \quad (B12)$$
$$Initp(0, m; 1, n) \mid Rule_{3-1} \quad (B13)$$
$$suc_{Control} = Result_1^{Ready} \mid Rule_{5-2} \quad (B14)$$
$$I = \rho^2{}_1 suc(I) \mid Rule_{5-3} \quad (B15)$$
$$Ready = eq?(I, m) \mid Rule_{5-4} \quad (B16)$$
$$Result = add(n, I, \rho^3{}_{Result_1}\#Q) \mid Rule_{5-5} \quad (B17)$$

- The inner-product cell code:

$$Result_1 = mux_1(arg_1, arg_2, arg_3, arg_4\#1) \mid Rule_2 \quad (B18)$$

$$Result_2 = mux_2(arg_1, arg_2, arg_3, arg_4\#1) \mid Rule_2 \quad (B19)$$
$$Result_3 = temp(Result_1, Result_2\#pro) \mid Rule_4 \quad (B20)$$
$$Result_4 = mux_3(arg_1, arg_2, arg_3, arg_4\#4) \mid Rule_2 \quad (B21)$$
$$Result_5 = add\_one(Result_1\#\lambda) \mid Rule_4 \quad (B22)$$
$$Initp(0, m; 1, n) \mid Rule_{5-1} \quad (B23)$$
$$suc_{Control} = Result_3^{Ready} \mid Rule_{5-2} \quad (B24)$$
$$I = \rho^2{}_1 suc(I) \mid Rule_{5-3} \quad (B25)$$
$$Ready = eq?(I, m) \mid Rule_{5-4} \quad (B26)$$
$$Result = inner\_product(n, I, \rho^3{}_{Result}\ \#add\_one) \mid Rule_{5-5} \quad (B27)$$

REFERENCES

1 **Sait, S M and H. Youssef, H** *VLSI Design Automation Theory and Practice*, McGraw Hill (1994)

2 **Falkoff, A D, Iverson, K E and Susseguth, E H** 'A formal description of system/360', *IBM System J.*, Vol 3 (1964) pp 198-262

3 **Hill, F J and Peterson, G R** *Digital Systems: Hardware Organization and Design*, John Wiley and Sons, New York (1973) (2nd ed. published 1978)

4 **Chu, Y** *Computer Organization and Microprogramming*, Prentice Hall, Englewood Cliffs, New York (1972)

5 **Duley, D R and Dietmeyer, D L** 'Translation of DDL digital system specification to boolean equations', *IEEE Trans. Computer*, Vol 18 (April 1969) pp 116-118

6 **Pilotly, R** 'Segmentation constructs for RTS III', *Proc. 1975 Int. Symposium on Hardware Description Languages*, New York, Vol 1 (September 1975) pp 115-124

7 **Barbacci, M R** 'Instruction Set Processor Specification (ISPS): The notation and its application', *IEEE Trans. Computer*, Vol 30 (January 1981) pp 24-40

8 **Sait, S M, Benten, M S T and Asjad Khan** 'Designing ASICs with UAHPL', *IEEE Circuits and Devices Magazine* (March 1995) pp 14-24

9 **Mesztenyi, K** 'Computer design language, simulation and boolean translation', Technical Report 68-72, Department of Computer Science, University of Maryland (June 1968)

10 **Knudsen, M J** 'PMSL, An Interactive Language for System-level Descripotion and Analysis of Computer Structures', PhD thesis, Carnegie Mellon University (1973)

11 **Dewey, A** 'The VHSIC Hardware Description Language', *VLSI Design* (1984)

12 **Tseng, C J** 'Automated synthesis of data paths in digital systems', PhD thesis, CMU (1984)

13 **Kowalski, T J** *et al.* 'The VLSI design automation assistant: from algorithms to silicon', *IEEE Design and Test* (August 1985)

14 **Granacki, J, Knapp, D and Parker, A** 'The ADAM advanced design automation system: Overview, planner and natural language interface', *2nd DAC*, ACM/IEEE (June 1085) pp 726-731

15 **Krekelberg, E, Shragowitz, E, Sobleman, G E and Lin-Shin Lin** 'Automated layout synthesis in the YASC silicon compiler', *2nd DAC* (June 1986) pp 447-453

16 **Marwedel, P** 'The MIMOLA design system: Tools for the design of digital processors', *Proc. 23rd ACM/IEEE Design Automation. Conf.* (1986) pp 587-593

17 **Terman, C** 'RSIM - A logic-level timing simulator', *Proc. Int. Conf. Computer Design* (October 1983) pp 437-440

18 **Trickey, H W** 'Compiling Pascal into Silicon', PhD thesis, Stanford University, CA (1985)

19 **Organick, E I** *et al.* 'Transforming an Ada programming unit to silicon and verifying its behavior in an Ada environment: A first experiment', *IEEE Software*, Vol 1 (January 1984) pp 31-49

20 **Girczyc, E F and Knight, J P** 'An Ada to standard cell hardware compiler based on graph grammars and scheduling', *ICCD'84*, New York (October 1984) pp 726-731

21 **Tanaka, T. Kobayashi, T and Karatsu, O** 'Harp: Fortran to silicon', *IEEE Trans. Computer Aided Design*, Vol 8 No 6 (June 1989) pp 8-14

22 **Gajski, D D** *Silicon Compilation*, Addison-Wesley, Reading, MA (1988)

23 **Johnson, S D** 'Synthesis of Digital Designs from Recursive Equations', PhD thesis, Computer Science Dept., Indiana University (1983)

24 **Ling, N and Bayoumi, M A** 'Temporal arithmetic: A new formalism for the specification and verification of systolic arrays', *IEEE Trans. Computer Aided Design*, Vol 9 No 8 (June 1990) pp 804-820

25 **Ling, N and Bayoumi, M A** 'Formal specification and verification strategies of systolic arrays', *Int. J. Computer Aided VLSI Design* (1991) pp 91-112

26 **Elleithy, K M and Bayoumi, M A** 'A framework for high level synthesis of digital architectures from μ-recursive algorithm', *Proc. ACM 18th Ann. Computer Science Conf.* (February 1990) pp 305-311

27 **Elleithy, K M** 'A Formal Framework For High Level Synthesis of Digital Designs', PhD thesis, The Center for Advanced Computer Studies, University of South-Western Louisiana, LA (1990)

28 **Bombana, M, Cavalloro, P and Zaza, G** 'Specification and formal synthesis of digital circuits', *Proc. IFIP TC10/WG10.2 Int. Workshop on Higher Order Logic Theorem Proving and its Applications, HDL '92* (1992)

29 **Kishinevsky, M A, Kondratyev, A Y and Taubin, A R** 'Formal methods for self-timed design', *Proc. Euro. Conf. Design Automation* (1991) pp 197-201

30 **Elleithy, E M and Bayoumi, M A** 'Synthesizing DSP architectures from behavioral specifications: A formal approach', *Proc. IEEE Int. Symposium on Circuits and Systems*, Vol 2 (May 1990) pp 1131-1134

31 **Masud-ul-Hasan** 'Back-End Design, of a Formal High Level Synthesis System', Master's thesis, King Fahd University of Petroleum and Minerals, Dhahran (1993)

32 *VLSI Design Tools Reference Manual*, NW Laboratory for Integrated Systems, release 3.1 edition (February 1987)

33 **Suaris, P R and Kedem, G** 'A new approach to standard cell layout', *Int. Conf. Computer Aided Design* (November 1987) pp 474-477

34 MCNC's Center for Microelectronics, *Open Architecture Silicon Implementation Software*, release 2.0 edition (December 1992)

35 **Mayo, R N, Arnold, M H, Scott, W S, Stark, D and Hamachi, G T** 'Decwrl/livermore magic release', Technical report, DECWRL, Digital Western Research

Laboratory (September 1990)

36 **Rose, J E** 'Greedy Algorithms for Wiring in VLSI', Master's thesis, Department of Computer Science, North Carolina State University (1985)

37 **Lorenzetti, M J, Nifog, M S and Rose, J E** 'Chann routing for compaction', *Proc. Int. Workshop on Plac ment and Routing* (May 1988)